CS 33

Signals Part 2

Signal Handlers and Masking

- **What if a signal occurs while a previous instance is being handled?**
  - **inconvenient …**
- **Signals are masked while being handled**
  - **may mask other signals as well:**

```
struct sigaction act; void myhandler(int);
sigemptyset(&act.sa_mask); // zeroes the mask
sigaddset(&act.sa_mask, SIGQUIT);
   // also mask SIGQUIT
act.sa_flags = 0;
act.sa_handler = myhandler;
sigaction(SIGINT, &act, NULL);
```


In the code example, we are setting up a handler for the SIGINT signal. SIGINT will automatically be masked while an occurrence of it is being handled, but the code arranges so that SIGQUIT is also masked while SIGINT is being handled.

Timed Out!

```
int TimedInput( ) {
   signal(SIGALRM, timeout);
   …
   alarm(30);     /* send SIGALRM in 30 seconds */
   GetInput();    /* possible long wait for input */
   alarm(0);      /* cancel SIGALRM request */
   HandleInput();
   return(0);
nogood:
   return(1);
}

void timeout( ) {
   goto nogood;   /* not legal but straightforward */
}
```


This slide sketches something that one might want to try to do: give a user a limited amount of time (in this case, 30 seconds — the **alarm** function causes the system to send the process a SIGALRM signal in the given number of seconds) to provide some input, then, if no input, notify the caller that there is a problem. Here we'd like our timeout handler to transfer control to someplace else in the program, but we can't do this.

Doing It Legally (but Weirdly)

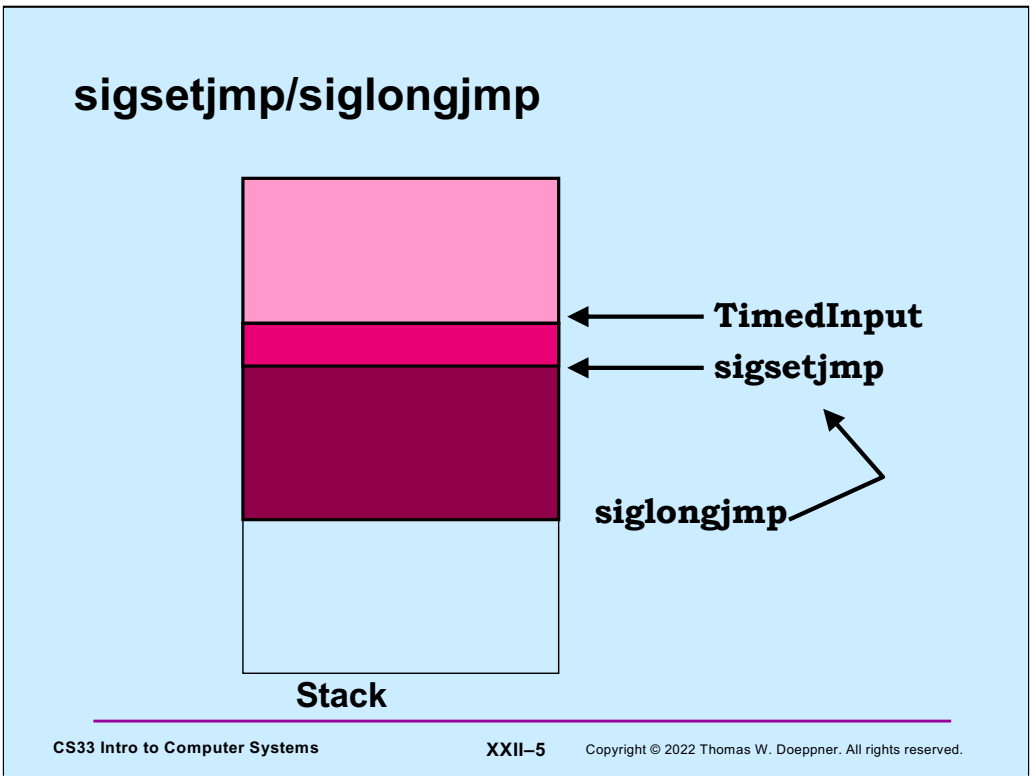
```
sigjmp_buf context;

int TimedInput( ) {
   signal(SIGALRM, timeout);
   if (sigsetjmp(context, 1) == 0) {
      alarm(30);  // cause SIGALRM in 30 seconds
      GetInput(); // possible long wait for input
      alarm(0);   // cancel SIGALRM request
      HandleInput();
      return 0;
   } else
      return 1;
}

void timeout() {
   siglongjmp(context, 1);    /* legal but weird */
}
```


To get around the problem of not being able to use a **goto** statement to get out of a signal handler, we introduce the **setjmp/longjmp** facility, also known as the ***nonlocal goto***. A call to **sigsetjmp** stores context information (about the current locus of execution) that can be restored via a call to **siglongjmp**. A bit more precisely: **sigsetjmp** stores into its first argument the values of the program-counter (instruction-pointer), stack-pointer, and other registers representing the process's current execution context. If the second argument is non-zero, the current signal mask is saved as well. The call returns 0. When **siglongjmp** is called with a pointer to this context information as its first argument, the current register values are replaced with those that were saved. If the signal mask was saved, that is restored as well. The effect of doing this is that the process resumes execution where it was when the context information was saved: inside of **sigsetjmp**. However, this time, rather than returning zero, it returns the second argument passed to **siglongjmp** (1 in the example).

To use this facility, you must include the header file **setjmp.h**.



In this slide, we start off in **TimedInput** and call **sigsetjmp**. The effect of **sigsetjmp** is to save the registers relevant to the current stack frame; in particular, the instruction pointer, the base pointer (if used), and the stack pointer, as well as the return address and the current signal mask. Thus when **sigsetjmp** returns, its context (including stack frame) is saved in the **jmpbuf**. **TimedInput** calls **GetInput** and its stack frame is pushed on the stack. If an alarm signal occurs, the stack frame for **timeout** is pushed on the stack. The call to **siglongjmp** (from within **Timeout**) restores the stack to what it was at the time of the call to **sigsetjmp** (including pushing its return address to **TimedInput** onto the stack). Thus the stack pointer register now points to the same location as it did when we first called sigsetjmp. and then **sigsetjmp** returns (again) to **TimedInput**, this time returning a different value than it did the first time.

Note that **siglongjmp** should be called only from a stack frame that is farther on the stack than the one in which **sigsetjmp** was called.

Job Control

```
$ who
```
  - foreground job

```
$ multiprocessProgram
```
  - foreground job

```
^Z
stopped
$ bg
[1] multiprocessProgram &
```
  - multiprocessProgram becomes background job 1

```
$ longRunningProgram &
[2]
$ fg %1
multiprocessProgram
```
  - multiprocessProgram is now the foreground job

```
^C
$
```

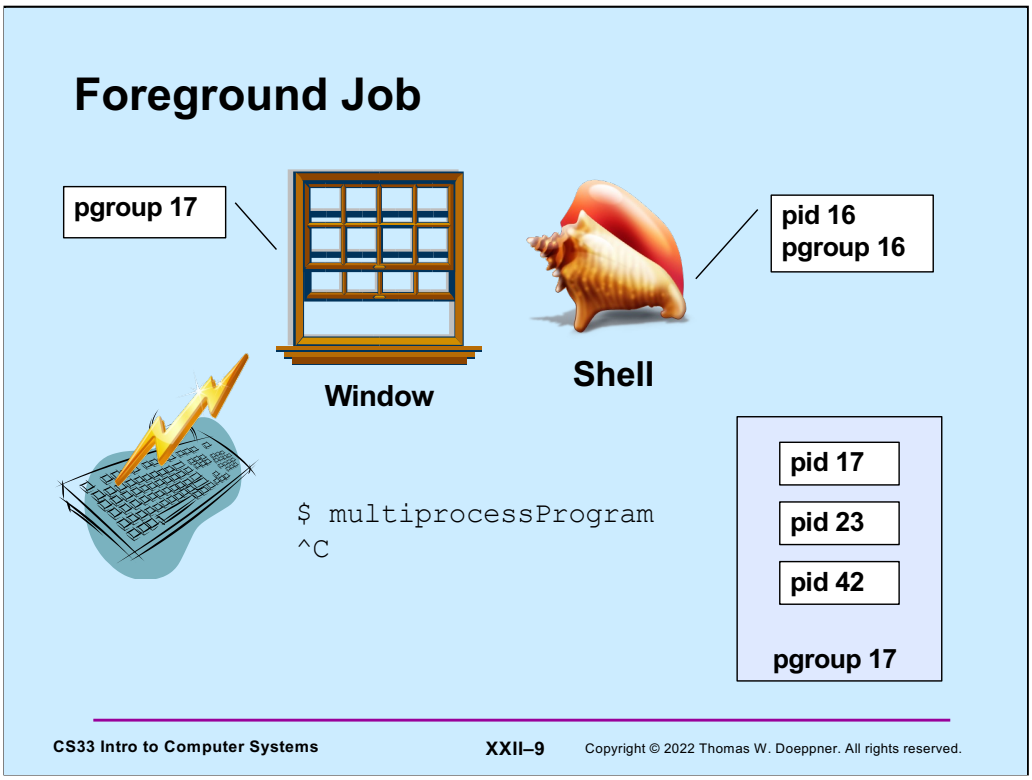
Process Groups

- Set of processes sharing the window/keyboard
  - sometimes called a *job*
- Foreground process group/job
  - currently associated with window/keyboard
  - receives keyboard-generated signals
- Background process group/job
  - not currently associated with window/keyboard
  - doesn't currently receive keyboard-generated signals

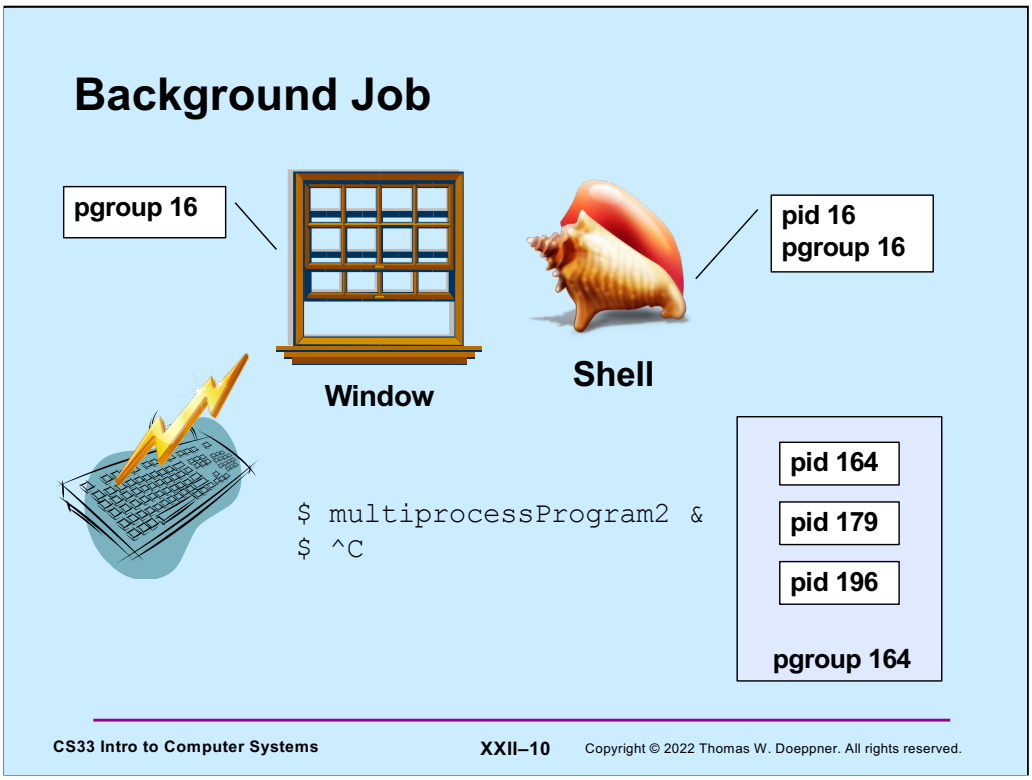
Keyboard-Generated Signals

- You type ctrl-C
- How does the system know which process(es) to send the signal to?

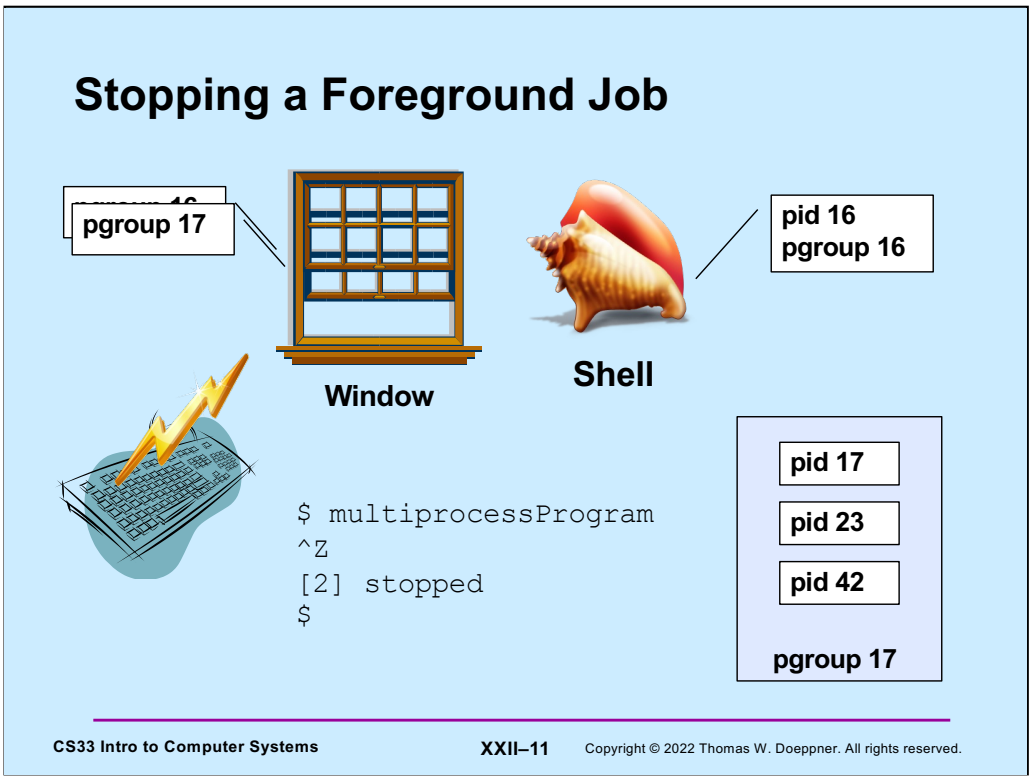
Each terminal window has a **process group** associated with it — this defines the current **foreground process group**. Keyboard-generated signals are sent to all processes in the current window's process group. This group normally consists of the shell and any of its descendents that have not been moved to other process groups.



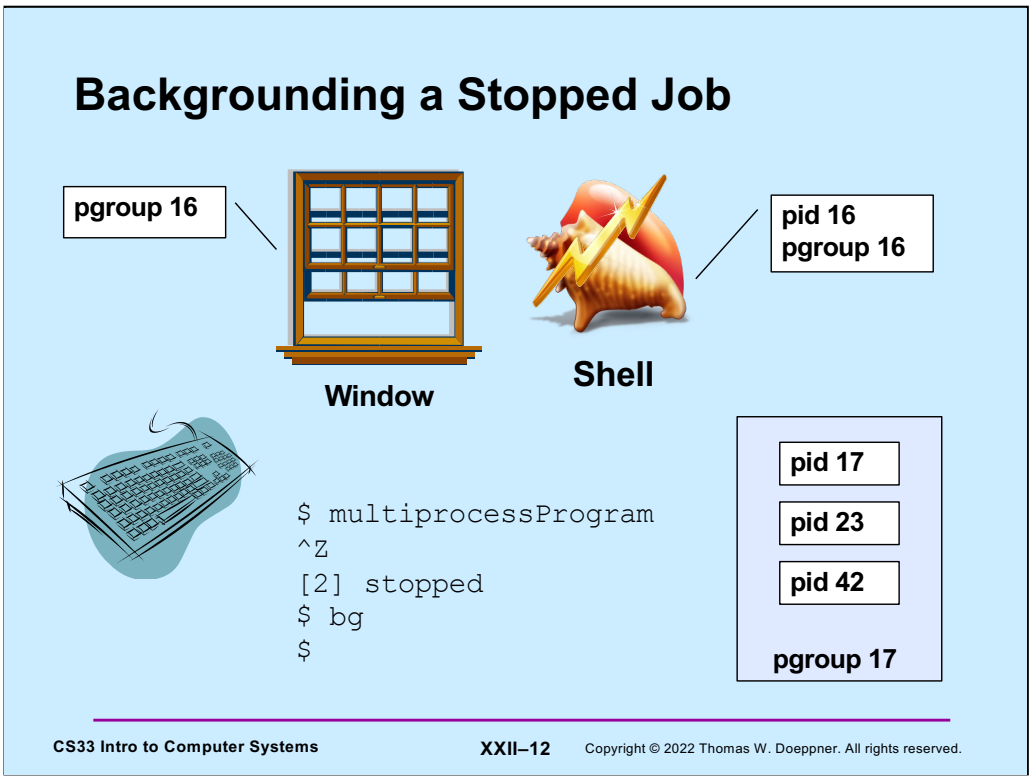
When you type a command into the shell without an ampersand, the shell makes sure that all the processes of that command are in a separate process group, shared with no other processes. The shell changes the window's process group to be that of the job, so that keyboard-generated signals are directed to the processes of the job and not to the shell. A process group's ID is the pid of its first member.



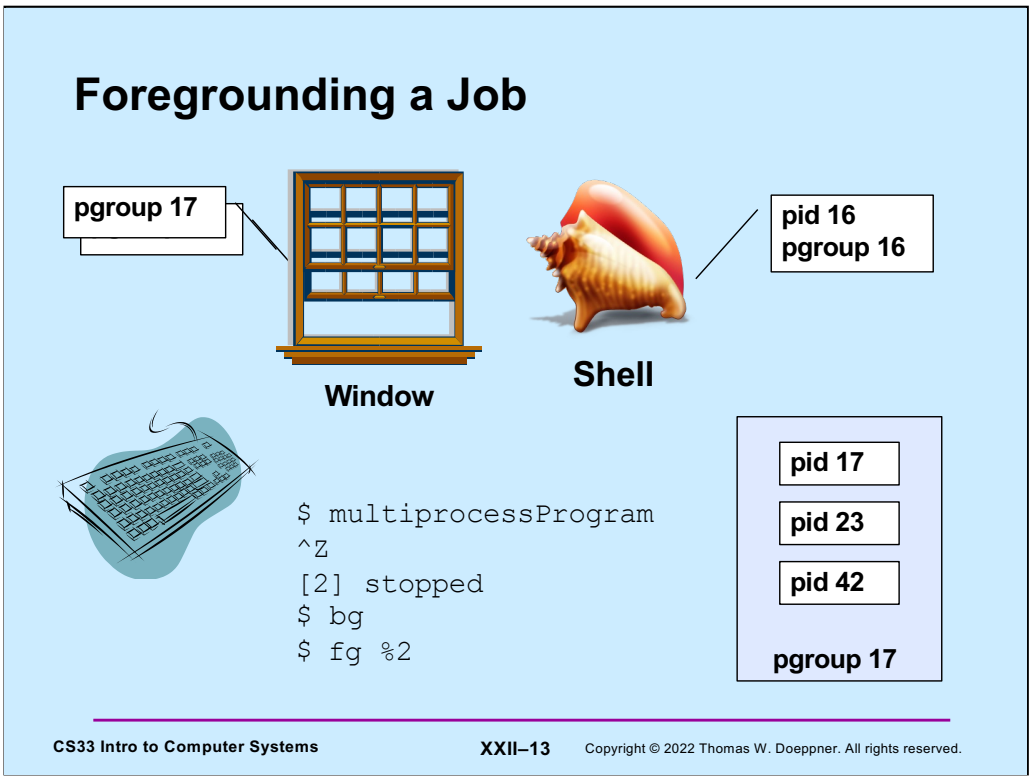
Keyboard-generated signals are not delivered to background jobs (for example, commands that are typed in with ampersands).



When you stop (or, synonymously, suspend) a foreground job, its execution is suspended (by sending it a SIGTSTP) and it is replaced as the foreground job by the shell.



If you then give the **bg** command to the shell, the most recently suspended job is sent a SIGCONT, which causes it to resume execution in the background, while the shell continues as the foreground job.



The **fg** command brings a job back to the foreground. Given with no arguments, the most recently suspended or backgrounded job is brought to the foreground, otherwise the argument specifies which job to bring to the foreground. In our example, the job that is being brought to the foreground is currently running in the background, so all that's necessary is for the shell to change the process group of the window and then wait for the job to terminate.

Quiz 1

```
$ long_running_prog1 &
$ long_running_prog2
^Z
[2] stopped
$ ^C
```

Which process group receives the SIGINT signal?

a) the one containing long_running_prog1
b) the one containing long_running_prog2
c) the one containing the shell

Creating a Process Group

```
if (fork() == 0) {
  // child
  setpgid(0, 0);
     /* puts current process into a
        new process group whose ID is
        the process's pid.
        Children of this process will be in
        this process's process group.
     */
  ...
  execv(...);
}
// parent
```


The first argument to **setpgid** is the process ID of the process whose process group is being changed; 0 means the **pid** of the calling process. The second argument is the ID of the **process group** it's being added to. If it's 0, then a new group is created whose ID is that of the calling process. Future children of this process join the new process group.

Setting the Foreground Process Group

```
tcsetpgrp(fd, pgid);
  // sets the process group of the
  // terminal (window) referenced by
  // file descriptor fd to be pgid
```

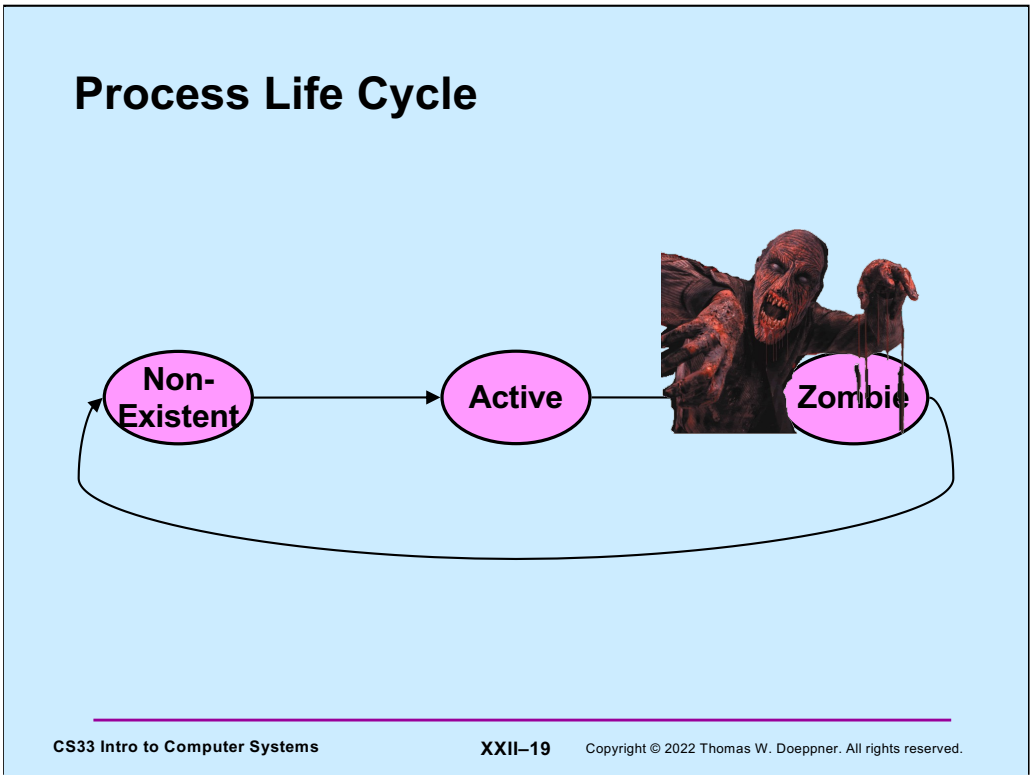

The **tcsetpgrp** command sets the process group associated with a terminal (i.e., a window), thus setting that process group to be the foreground process group.

Background Input and Output

- Background process reads from keyboard
  - the keyboard really should be reserved for foreground process
  - background process gets SIGTTIN
    - » suspends it by default
- Background process writes to display
  - display also used by foreground process
  - could be willing to share
  - background process gets SIGTTOU
    - » suspends it (by default)
    - » but reasonable to ignore it

Kill: Details

- `int kill(pid_t pid, int sig)`
  - if *pid* > 0, signal *sig* sent to process *pid*
  - if *pid* == 0, signal *sig* sent to all processes in the caller's process group
  - if *pid* == −1, signal *sig* sent to all processes in the system for which sender has permission to do so
  - if *pid* < −1, signal *sig* is sent to all processes in process group −*pid*



A Unix process is always in one of three states, as shown in the slide. When created, the process is put in the **active** state. When a process terminates, its parent might wish to find out and, perhaps, retrieve the exit value. Thus when a process terminates, some information about it must continue to exist until passed on to the parent (via the parent's executing the **wait** or **waitpid** system call). So, when a process calls **exit**, it enters the **zombie** state and its exit code is kept around. Furthermore, the process's ID is preserved so that it cannot be reused by a new process. Once the parent does its **wait**, the exit code and process ID are no longer needed, so the process completely disappears and is marked as being in the **non-existent** state — it doesn't exist anymore. The process ID may now be reused by a new process.

Reaping: Zombie Elimination

- **Shell must call `waitpid` on each child**
  - **easy for foreground processes**
  - **what about background?**

```
pid_t waitpid(pid_t pid, int *status, int options);
```

  - ***pid* values:**

< −1	**any child process whose process group is |pid|**
−1	**any child process**
0	**any child process whose process group is that of caller**
> 0	**child process whose ID is equal to pid**

  - `wait(&status)` **is equivalent to** `waitpid(-1, &status, 0)`

A process may wait only for its children to terminate (this excludes grandchildren).

(continued)

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- ***options*** **are some combination of the following**
  - » **WNOHANG**
    - **return immediately if no child has exited (returns 0)**
  - » **WUNTRACED**
    - **also return if a child has been stopped (suspended)**
  - » **WCONTINUED**
    - **also return if a child has been continued (resumed)**

If a process is found, **waitpid** returns the process ID of the process that has been suspended or resumed.

When to Call `waitpid`

- Shell reports status only when it is about to display its prompt
  - thus sufficient to check on background jobs just before displaying prompt

`waitpid` status

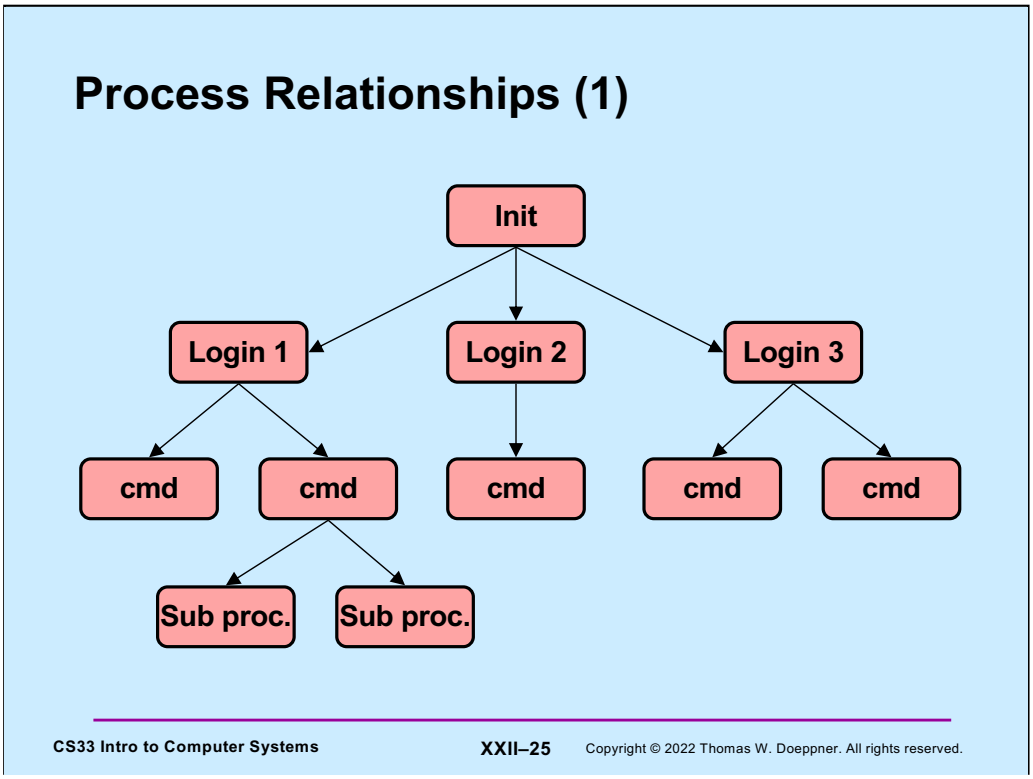
- WIFEXITED(*status): 1 if the process terminated normally and 0 otherwise
- WEXITSTATUS(*status): argument to exit
- WIFSIGNALED(*status): 1 if the process was terminated by a signal and 0 otherwise
- WTERMSIG(*status): the signal which terminated the process if it terminated by a signal
- WIFSTOPPED(*status): 1 if the process was stopped by a signal
- WSTOPSIG(*status): the signal which stopped the process if it was stopped by a signal
- WIFCONTINUED(*status): 1 if the process was resumed by SIGCONT and 0 otherwise

These are macros that can be applied to the status output argument of **waitpid**. Note that "terminated normally" means that the process terminated by calling **exit**. Otherwise, it was terminated because it received a signal, which it neither ignored nor had a handler for, whose default action was termination.

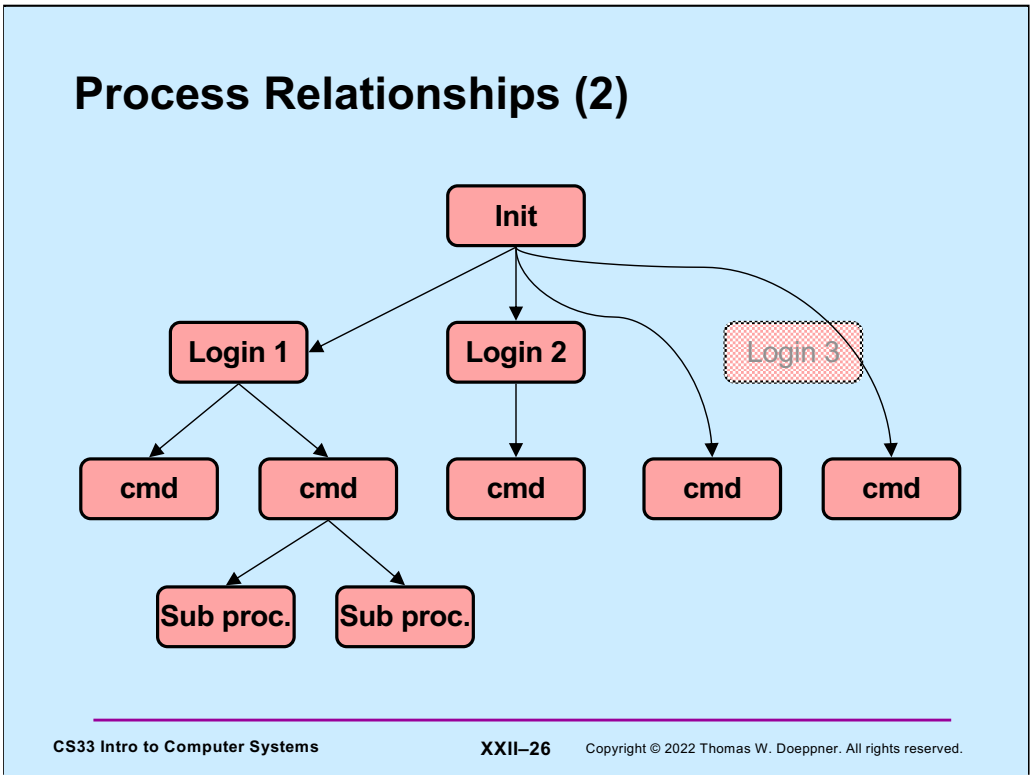
Example (in Shell)

```
int wret, wstatus;
while ((wret = waitpid(-1, &wstatus, WNOHANG|WUNTRACED)) > 0){
  // examine all children who've terminated or stopped
  if (WIFEXITED(wstatus)) {
    // terminated normally
    ...
  }
  if (WIFSIGNALED(wstatus)) {
    // terminated by a signal
    ...
  }
  if (WIFSTOPPED(wstatus)) {
    // stopped
    ...
  }
}
```


This code might be executed by a shell just before it displays its prompt. The loop iterates through all child processes that have either terminated or stopped. The WNOHANG option causes **waitpid** to return 0 (rather than waiting) if the caller has extant children, but there are no more that have either terminated or stopped. If the caller has no children, then **waitpid** returns -1.

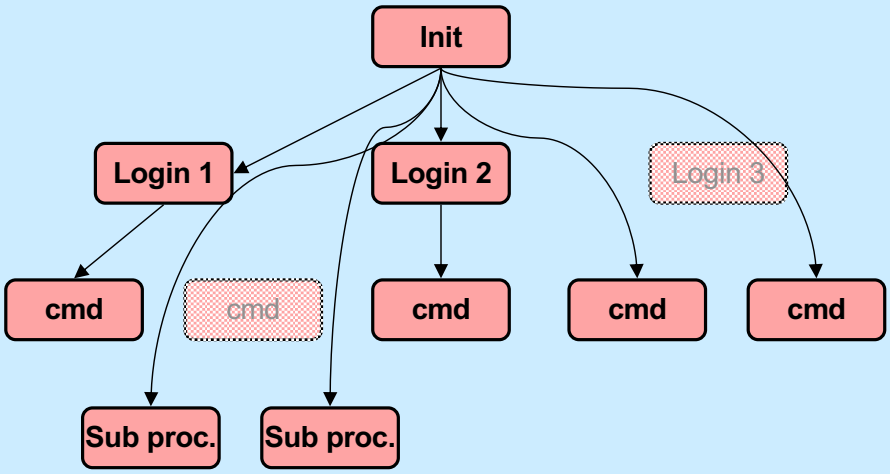


The **init** process is the common ancestor of all other processes in the system. It continues to exist while the system is running. It starts things going soon after the system is booted by forking child processes that exec the login code. These login processes then exec the shell. Note that, since only the parent may wait for a child's termination, only parent-child relationships are maintained between processes.



When a process terminates, all of its children are inherited by the **init** process, process number 1.

Process Relationships (3)



Signals, Fork, and Exec

```
// set up signal handlers ...
if (fork() == 0) {
   // what happens if child gets signal?
   ...
   signal(SIGINT, SIG_IGN);
   signal(SIGFPE, handler);
   signal(SIGQUIT, SIG_DFL);
   execv("new prog", argv, NULL);
   // what happens if SIGINT, SIGFPE,
   // or SIGQUIT occur?
}
```


As makes sense, the signal-handling state of the parent is reproduced in the child.

What also makes sense is that, if a signal has been given a handler, then, after an **exec**, since the handler no longer exists, the signal reverts to default actions.

What at first glance makes less sense is that ignored signals stay ignored after an **exec** (of course, signals with default action stay that way after the **exec**). The intent is that this allows one to run a program protected from certain signals.

Signals and System Calls

- What happens if a signal occurs while a process is doing a system call?
  - handler not invoked until just before system call returns to user
    - » system call might terminate early because of signal
  - system call completes
  - signal handler is invoked
  - user code resumed as if the system call has just returned

It's generally unsafe to interrupt the execution of a process while it's in the midst of doing a system call. Thus, if a signal is sent to a process while it's in a system call, it's usually not acted upon until just before the process returns from the system call back to the user code. At this point the handler (if any) is executed. When the handler returns, normal execution of the the user process resumes and it returns from the system call.

Signals and Lengthy System Calls

- Some system calls take a long time
  - large I/O transfer
    - » multi-gigabyte read or write request probably done as a sequence of smaller pieces
  - a long wait is required
    - » a read from the keyboard requires waiting for someone to type something
- If signal arrives in the midst of lengthy system call, handler invoked:
  - after current piece is completed
  - after cancelling wait

Some system calls take a long time to execute. Such calls might be broken up into a sequence of discrete steps, where it's safe to check for and handle signals after each step. For example, if a process is writing multiple gigabytes of data to a file in a single call to **write**, the kernel code it executes will probably break this up into a number of smaller writes, done one at a time. After each write completes, it checks to see if any unmasked signals are pending.

Interrupted System Calls

- **What if a signal is handled before the system call completes?**
  - **invoke handler, then return from system call prematurely**
    - **if one or more pieces were completed, return total number of bytes transferred**
    - **otherwise return "interrupted" error**

What happens to the system call after the signal handling completes (assuming that the process has not been terminated)? The system call effectively terminated when the handler was called. When the handler returns, the system call either returns an indication of how far it progressed before being interrupted by the signal (it would return the number of bytes actually transferred, as opposed to the number of bytes requested) or, if it was interrupted before anything actually happened, it returns an error indication and sets **errno** to EINTR (meaning "interrupted system call").

Interrupted System Calls: Non-Lengthy Case

```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call — try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```


If a non-lengthy system call is interrupted by a signal, the call fails and the error code EINTR is put in **errno**. The process then executes the signal handler and then returns to the point of the interrupt, which causes it to (finally) return from the system call with the error.

Quiz 2

```
int ret;
char buf[1024*1024*1024];

fillbuf(buf);

ret = write(1, buf, 1024*1024*1024);
```

- **The value of ret is:**
  - **a) any integer in the range [-1, 1024*1024*1024]**
  - **b) either -1 or 1024*1024*1024**
  - **c) either -1, 0, or 1024*1024*1024**

Interrupted System Calls: Lengthy Case

```
char buf[BSIZE];
fillbuf(buf);
long remaining = BSIZE;
char *bptr = buf;
while (1){
  long num_xfrd = write(fd,
       bptr, remaining);
  if (num_xfrd == -1) {
    if (errno == EINTR) {
      // interrupted early
      continue;
    }
    perror("big trouble");
    exit(1);
  }
  if (num_xfrd < remaining) {
    /* interrupted after the
       first step */
    remaining -= num_xfrd;
    bptr += num_xfrd;
    continue;
  }
  // success!
  break;
}
```


The actions of some system calls are broken up into discrete steps. For example, if one issues a system call to write a gigabyte of data to a file, the write will actually be split by the kernel into a number of smaller writes. If the system call is interrupted by a signal after the first component of the write has completed (but while there are still more to be done), it would not make sense for the call to return an error code: such an error return would convince the program that none of the write had completed and thus all should be redone. Instead, the call completes successfully: it returns the number of bytes actually transferred, the signal handler is invoked, and, on return from the signal handler, the user program receives the successful return from the (shortened) system call.

Asynchronous Signals (1)

```
main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...   /* long-running buggy code */

}

void handler(int sig) {
   ...   /* clean up */
   exit(1);
}
```


Let's look at some of the typical uses for asynchronous signals. Perhaps the most common is to force the termination of the process. When the user types control-C, the program should terminate. There might be a handler for the signal, so that the program can clean up and then terminate.

Asynchronous Signals (2)

```
computation_state_t  state;

main( ) {
  void handler(int);

  signal(SIGINT, handler);

  long_running_procedure( );
}
```

```
long_running_procedure( ) {
  while (a_long_time) {
    update_state(&state);
    compute_more( );
  }
}

void handler(int sig) {
  display(&state);
}
```


Here we are using a signal to send a request to a running program: when the user types control-C, the program prints out its current state and then continues execution. If synchronization is necessary so that the state is printed only when it is stable, it must be provided by appropriate settings of the signal mask.

Asynchronous Signals (3)

```
main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ... /* complicated program */

  myputs("important message\n");

  ... /* more program */

}
```

```
void handler(int sig) {

  ... /* deal with signal */

  myputs("equally important "
      "message\n");
}
```


In this example, both the mainline code and the signal handler call **myputs**, which is similar to the standard-I/O routine *puts*. It's possible that the signal invoking the handler occurs while the mainline code is in the midst of the call to **myputs**. Could this be a problem?

Asynchronous Signals (4)

```
char buf[BSIZE];
int pos;
void myputs(char *str) {
  int len = strlen(str);
  for (int i=0; i<len; i++, pos++) {
    buf[pos] = str[i];
    if ((buf[pos] == '\n') || (pos == BSIZE-1)) {
      write(1, buf, pos+1);
      pos = -1;
    }
  }
}
```


Here's the implementation of **myputs**, used in the previous slide. What it does is copy the input string, one character at a time, into **buf**, which is of size BSIZE. Whenever a newline character is encountered, the current contents of **buf** up to that point are written to standard output, then subsequent characters are copied starting at the beginning of **buf**. Similarly, if **buf** is filled, its contents are written to standard output and subsequent characters are copied starting at the beginning of **buf**. Since **buf** is global, characters not written out may be written after the next call to **myput**. Note that **printf** (and other stdio routines) buffers output in a similar way.

The point of **myputs** is to minimize the number of calls to *write*, so that **write** is called only when we have a complete line of text or when its buffer is full.

However, consider what happens if execution is in the middle of **myputs** when a signal occurs, as in the previous slide. Among the numerous problem cases, suppose **myput** is interrupted just after **pos** is set to -1 (if the code hadn't had been interrupted, **pos** would be soon incremented by 1). The signal handler now calls **myputs**, which copies the first character of **str** into **buf[pos],** which, in this case, is **buf[-1].** Thus the first character "misses" the buffer. At best it simply won't be printed, but there might well be serious damage done to the program.

Async-Signal Safety

- **Which library functions are safe to use within signal handlers?**

– abort	– dup2	– getppid	– readlink	– sigemptyset	– tcgetpgrp
– accept	– execle	– getsockname	– recv	– sigfillset	– tcsendbreak
– access	– execve	– getsockopt	– recvfrom	– sigismember	– tcsetattr
– aio_error	– _exit	– getuid	– recvmsg	– signal	– tcsetpgrp
– aio_return	– fchmod	– kill	– rename	– sigpause	– time
– aio_suspend	– fchown	– link	– rmdir	– sigpending	– timer_getoverrun
– alarm	– fcntl	– listen	– select	– sigprocmask	– timer_gettime
– bind	– fdatasync	– lseek	– sem_post	– sigqueue	– timer_settime
– cfgetispeed	– fork	– lstat	– send	– sigsuspend	– times
– cfgetospeed	– fpathconf	– mkdir	– sendmsg	– sleep	– umask
– cfsetispeed	– fstat	– mkfifo	– sendto	– sockatmark	– uname
– cfsetospeed	– fsync	– open	– setgid	– socket	– unlink
– chdir	– ftruncate	– pathconf	– setpgid	– socketpair	– utime
– chmod	– getegid	– pause	– setsid	– stat	– wait
– chown	– geteuid	– pipe	– setsockopt	– symlink	– waitpid
– clock_gettime	– getgid	– poll	– setuid	– sysconf	– write
– close	– getgroups	– posix_trace_event	– shutdown	– tcdrain	
– connect	– getpeername	– pselect	– sigaction	– tcflow	
– creat	– getpgrp	– raise	– sigaddset	– tcflush	
– dup	– getpid	– read	– sigdelset	– tcgetattr	

To deal with the problem on the previous page, we must arrange that signal handlers cannot destructively interfere with the operations of the mainline code. Unless we are willing to work with signal masks (which can be expensive), this means we must restrict what can be done inside a signal handler. Routines that, when called from a signal handler, do not interfere with the operation of the mainline code, no matter what that code is doing, are termed **async-signal safe**. The POSIX 1003.1 spec requires the functions shown in the slide to be async-signal safe.

Note that POSIX specifies only those functions that must be async-signal safe. Implementations may make other functions async-signal safe as well.

Quiz 3

Printf is not listed as being async-signal safe. Can it be implemented so that it is?

a) yes, but it would be so complicated, it's not done
b) yes, it can be easily made async-signal safe
c) no, it's inherently not async-signal safe